Process Creation

Events which cause process creation:

- System initialization.
- Execution of a process creation system call by a running process.
  - In Linux/UNIX: fork()
  - In Windows CreateProcess()
- A user request to create a new process.
- Initiation of a batch job.
Process Termination

Events which cause process termination:

- Normal exit (voluntary).
  - Using C call `exit(0);`
- Error exit (voluntary).
  - Using C call `exit(N);` where $0 < N < 256$ in Linux
- Fatal error (involuntary).
  - Process receives a signal in Linux/UNIX
- Killed by another process (involuntary).
  - Process receives a signal in Linux/UNIX
Process Components

Major Components of a Linux/UNIX Process

- PID
- PPID
- UID RUID and EUID
- GID RGID and EGID
- Address Space (Minimum: TEXT, GLOBAL DATA, STACK)
- Executable Program
- One or more Threads
- Default (Initial Thread) Scheduling Policy and Priority
- Current Working Directory
- Open Channel Table
- Signal Table
Address Space Model

N Byte Address Space

Addr 0

TEXT

GLOBAL DATA

STACK

Addr N - 1
### Implementation of Processes

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment info</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment info</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment info</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2-4. Some of the fields of a typical process table entry.*
Interrupts on a Process Thread

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.
Thread States

- fork()
- PREEMPT
- SLEEP
- block
- ready
- WAKEUP
- dispatch
- run
- k/u
- exit

fork()
Thread Usage (1)

Figure 2-7. A word processor with three threads.
The Classical Thread Model (1)

Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.
The Classical Thread Model (2)

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.
The Classical Thread Model (3)

Figure 2-13. Each thread has its own stack.
Critical Regions (1)

Conditions required to avoid race condition:

• No two threads may be simultaneously inside their critical regions. *(Mutex Requirement)*
• No assumptions may be made about speeds or the number of CPUs.
• No thread running outside its critical region may block other thread. *(Progress Requirement)*
• No thread should have to wait forever to enter its critical region. *(Bounded Waiting Requirement)*
Critical Regions (2)

Figure 2-22. Mutual exclusion using critical regions.
Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction
Figure 2-23. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.
Peterson's Solution

```c
#define FALSE 0
#define TRUE  1
#define N     2           /* number of processes */

int turn;            /* whose turn is it? */
int interested[N];   /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;        /* number of the other process */

    other = 1 - process;   /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;       /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Figure 2-24. Peterson’s solution for achieving mutual exclusion.
enter_region:
  MOVE REGISTER,#1
  XCHG REGISTER,LOCK
  CMP REGISTER,#0
  JNE enter_region
  RET

| put a 1 in the register  |
| swap the contents of the register and lock variable |
| was lock zero? |
| if it was non zero, lock was set, so loop |
| return to caller; critical region entered |

leave_region:
  MOVE LOCK,#0
  RET

| store a 0 in lock |
| return to caller |

Figure 2-26. Entering and leaving a critical region using the x-86 XCHG instruction.
Semaphores

- Basically an unsigned counter and a queue
- Two basic operations defined:
  - `wait(sem_object);` also `down();`, `p()`
  - `signal(sem_object);` also `up();`, `v()`
- A wait call is a conditional decrement
  - If sem counter is +, decrement and return
  - If sem counter is 0, block caller
- A signal call is a conditional increment
  - If no waiters, increment counter
  - If waiters, move one waiter to ready Q
GLOBAL TO PRODUCER AND CONSUMER THREADS:

```c
sem_t prod = 10;         sem_t cons = 0;
sem_t iptr = 1;            sem_t optr = 1;
```

```c
int buf[10], in=0, out=0;
void p ( sem_t * );
void v ( sem_t * );
```

**PRODUCER FUNCTION**

```c
void producer(){
    while(1){
        p(&prod);
        p(&iptr);
        buf[in] = random();
        in = (in + 1) % 10;
        v(&iptr);
    }
}
```

**CONSUMER FUNCTION**

```c
void consumer(){
    int val;
    while(1){
        p(&cons);
        p(&optr);
        val = buf[out];
        // print val somewhere
        out = (out + 1) % 10;
        v(&optr);
        v(&prod);
    }
}
```
Event Counters and Sequencers

- Semaphores may provide more functionality than needed to resolve certain kinds of synchronization requirements
  - Total order problems like the multiple producer / multiple consumer problem need the power of semaphores
  - Partial order problems like the single producer / single consumer problem do not need all of the functionality of a semaphore
- Event Counters can solve partial order problems more efficiently than semaphores
- Event Counters in conjunction with Sequencers can solve total order problems as efficiently as semaphores, and can provide additional functionality
Event Counters

• Basically an unsigned counter and a queue
• Two basic operations defined:
  • await(EventCounter, value);
  • advance (EventCounter);
• An await call is a test between EC and value
  • If value is <= EC return to caller
  • If value is > EC block caller
• An advance call is an unconditional EC increment
  • If any waiter has value <= EC after increment, then move such waiter(s) to ready Q
GLOBAL TO PRODUCER AND CONSUMER THREADS:
ec_t pEC, cEC;
int ring_buf[10];
unsigned in=0, out=0;
void await (ec_t *, int);
void advance (ec_t *);

PRODUCER FUNCTION

```c
void producer(){
    while(1){
        await(&pEC, in - 10 + 1);
        ring_buf[in % 10] = random();
        in = (in + 1);
        advance(&cEC)
    }
}
```

CONSUMER FUNCTION

```c
void consumer(){
    int val;
    while(1){
        await(&cEC, out + 1);
        val = ring_buf[out % 10];
        // print val somewhere
        out = (out + 1);
        advance(&pEC);
    }
}
```
Sequencers

- Basically an unsigned atomic counter
- One operation defined:
  - ticket(Sequencer);
- A ticket call atomically returns the next Sequencer value, and this value is generally used in an await(EC, ticket(Seq)) form of call
- Sequencers, in conjunction with Event Counters provide all of the synchronization capabilities of semaphores
GLOBAL TO PRODUCER AND CONSUMER THREADS:
ec_t pEC, cEC;
seq_t ps, cs;
int ring_buf[10];
unsigned in=0, out=0;
void await (ec_t * , int);
void advance (ec_t *);
int ticket (seq_t *);

PRODUCER FUNCTION

void producer(){
int t; // local to each pro
while(1){
t = ticket(&ps);
await(&cEC, t);
await(&pEC, t - 10 + 1);
ring_buf[t % 10] = random();
advance(&cEC)
}

CONSUMER FUNCTION

void consumer(){
int u, val; // local to each con
while(1){
u = ticket(&cs);
await(&pEC, u);
await(&cEC, u + 1);
val = ring_buf[u % 10];
// print val somewhere
advance(&pEC);
}
Monitors (1)

monitor example
    integer i;
    condition c;

procedure producer( );
    :
    :
end;

procedure consumer( );
    :
    :
end;
end monitor;

Figure 2-33. A monitor.
Monitors (2)

```
monitor ProducerConsumer
condition full, empty;
integer count;

procedure insert(item: integer);
begin
  if count = N then wait(full);
  insert_item(item);
  count := count + 1;
  if count = 1 then signal(empty)
end;

function remove: integer;
begin
  if count = 0 then wait(empty);
  remove = remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
end:

  count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors.
MULTIPLE PRODUCER, MULTIPLE CONSUMER RING BUFFER EXAMPLE
USING A MONITOR IN THE LANGUAGE CSP/k

01 CIRCULARBUFFER: PROCEDURE OPTIONS (CONCURRENT);
02
03 CIRCULARBUFFERMONITOR: MONITOR;
04    DECLARE (BUFFERS (100)) CHARACTER (80) VARYING;
05    DECLARE (FIRSTBUFFER, LASTBUFFER) FIXED;
06    DECLARE (TOTALBUFFERS, FULLBUFFERS) FIXED;
07    DECLARE (ABUFFERISEMPTY) CONDITION;
08    DECLARE (ABUFFERISFULL) CONDITION;
09
10 DO;
11    FIRSTBUFFER = 1;
12    LASTBUFFER = 1;
13    TOTALBUFFERS = 100;
14    FULLBUFFERS = 0;
15 END;
16
17 SPOOLER: ENTRY (IMAGE);
18    DECLARE (IMAGE) CHARACTER (*) VARYING;
19    IF FULLBUFFERS = TOTALBUFFERS THEN
20        WAIT (ABUFFERISEMPTY);
21        BUFFERS (LASTBUFFER) = IMAGE;
22        LASTBUFFER = MOD (LASTBUFFER, TOTALBUFFERS) + 1;
23        FULLBUFFERS = FULLBUFFERS + 1;
24        SIGNAL (ABUFFERISFULL);
25 END;
26
27 DESPOOLER: ENTRY (IMAGE);
28    DECLARE (IMAGE) CHARACTER (*) VARYING;
29    IF FULLBUFFERS = 0 THEN
30        WAIT (ABUFFERISFULL);
31        IMAGE = BUFFERS (FIRSTBUFFER);
32        FIRSTBUFFER = MOD (FIRSTBUFFER, TOTALBUFFERS) + 1;
33        FULLBUFFERS = FULLBUFFERS - 1;
34        SIGNAL (ABUFFERISEMPTY);
35 END;
36
37 END;
MULTIPLE PRODUCER, MULTIPLE CONSUMER RING BUFFER EXAMPLE USING A MONITOR IN THE LANGUAGE CSP/k (cont’d)

39 READCARDS: PROCESS;
40 DECLARE (CARDIMAGE) CHARACTER (80) VARYING;
41 CARDIMAGE = ‘MORECARDS’;
42 DO WHILE (CARDIMAGE <> ‘ENDOFFILE’);
43 GET SKIP EDIT (CARDIMAGE) (A(80));
44 CALL SPOOLER (CARDIMAGE);
45 END;
46 END;
47
48 PRINTLINES: PROCESS;
49 DECLARE (LINEIMAGE) CHARACTER (80) VARYING;
50 LINEIMAGE = ‘MORECARDS’;
51 DO WHILE (LINEIMAGE <> ‘ENDOFFILE’);
52 CALL DESPOOLER (LINEIMAGE);
53 PUT SKIP EDIT (LINEIMAGE) (A(80));
54 END;
55 END;
56
57 END;

CSP/k program for managing a circular buffer.
Mutexes

mutex_lock:
  TSL REGISTER,MUTEX                        | copy mutex to register and set mutex to 1
  CMP REGISTER,#0                           | was mutex zero?
  JZE ok                                    | if it was zero, mutex was unlocked, so return
  CALL thread_yield                         | mutex is busy; schedule another thread
  JMP mutex_lock                             | try again
  ok: RET                                   | return to caller; critical region entered

mutex_unlock:
  MOVE MUTEX,#0                             | store a 0 in mutex
  RET                                       | return to caller

Figure 2-29. Implementation of mutex lock and mutex unlock.
## Mutexes in Pthreads (1)

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_mutex_init</code></td>
<td>Create a mutex</td>
</tr>
<tr>
<td><code>pthread_mutex_destroy</code></td>
<td>Destroy an existing mutex</td>
</tr>
<tr>
<td><code>pthread_mutex_lock</code></td>
<td>Acquire a lock or block</td>
</tr>
<tr>
<td><code>pthread_mutex_trylock</code></td>
<td>Acquire a lock or fail</td>
</tr>
<tr>
<td><code>pthread_mutex_unlock</code></td>
<td>Release a lock</td>
</tr>
</tbody>
</table>

*Figure 2-30. Some of the Pthreads calls relating to mutexes.*
## Mutexes in Pthreads (2)

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>Pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>Pthread_cond_wait</td>
<td>Block waiting for a signal</td>
</tr>
<tr>
<td>Pthread_cond_signal</td>
<td>Signal another thread and wake it up</td>
</tr>
<tr>
<td>Pthread_cond_broadcast</td>
<td>Signal multiple threads and wake all of them</td>
</tr>
</tbody>
</table>

**Figure 2-31.** Some of the Pthreads calls relating to condition variables.
Figure 2-32. Using threads to solve the producer-consumer problem.
Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.
Categories of Scheduling Algorithms

- Batch
- Interactive
- Real time
Scheduling Algorithm Goals

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users’ expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.
Scheduling Parameters

- When a thread is created it is allocated a set of scheduling parameters
  - A scheduling policy
    - Batch, timeshare, real-time
  - A priority within that policy
    - Batch priorities are low, timeshare intermediate, real-time high
  - A possible time-slice (quantum)
    - Timeshare and real-time round robin use timeouts
  - Possible processor (core) affinity
    - A thread can be connected to one or a set of cores
  - Possible memory affinity
    - In NUMA systems, a thread can be connected to one or a set of cores that are closer to some specific part of RAM
  - Possible IO (bridge) affinity
    - In NUMA systems, a thread can be connected to one or a set of cores that are closer to some specific IO bridge
Buses

The bus structure of a pre-Nehalem Pentium 4
Enterprise: 2008 Nehalem Based
Two Socket System Architecture

Nehalem-EP Platform:
- Two sockets each with Integrated Memory Controller
- Turbo mode operation
- Intel® QuickPath Architecture
- DDR3 Memory: 3 Channel, 3 DIMMs per channel
- Intel® Virtualization Technology
- PCI Express® Gen 2

PCI Express® Gen 2
X58 I/O Hub
ICH 9/10
Intel® QuickPath Interconnect
Enterprise: 2009 Nehalem Based
Four Socket System Architecture

Boxboro-EX Platform:
- Four processors with Intel QuickPath Interconnects
- PCI Express Gen 2, Integrated Memory Controller

* Other names and brands may be claimed as the property of others
sched_setscheduler() sets both the scheduling policy and the associated parameters for the thread whose ID is specified in arg tid. If tid equals zero, the scheduling policy and parameters of the calling thread will be set. The interpretation of the argument param depends on the selected policy. Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies:

SCHED_OTHER  the standard round-robin time-sharing policy;
SCHED_BATCH   for "batch" style execution of processes; and
SCHED_IDLE    for running very low priority background jobs.

The following "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable threads are selected for execution:

SCHED_FIFO    a first-in, first-out policy; and
SCHED_RR      a round-robin policy.

Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling
Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.
Priority Scheduling

Figure 2-42. A scheduling algorithm with four priority classes.
Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.
Figure 2-43. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Possible:  A1, A2, A3, A1, A2, A3
Not possible:  A1, B1, A2, B2, A3, B3

Also possible:  A1, B1, A2, B2, A3, B3
Scheduling in Real Time Systems

- Real Time Issues
- FIFO RT
- RR RT
- Deadline Scheduling
Scheduling in Real Time Systems (2)

- Real Time Issues
  - Deterministic latency
    - Policies that can guarantee a minimum time bound from ready state to run state
  - Priority range
    - Generally higher than non RT policies
- Dynamic priority adjustment
  - Hands-off for all but deadline
FIFO Real Time Policy

- Highest Priority First (no RR)
- Once an HPF thread reaches the run state it cannot be preempted by another thread of the same highest priority
  - Run state is left only by EXIT, BLOCK operation or Priority Preemption (no RR)
  - Another thread of the same priority can only run when the first FIFO thread leaves the run state
Round Robin Real Time Policy

- Highest Priority First with RR
- Once an HPF thread reaches the run state it can be preempted by another thread of the same highest priority when its quantum expires
  - Run state is left by EXIT, BLOCK operation, Quantum Expiration or Priority Preemption
  - Another thread of the same priority can run if first RR thread completes its time slice
Deadline Real Time Policy

• A thread’s priority is dynamically adjusted as the thread approaches a predetermined deadline
• The intent is to make sure that the deadline scheduled thread will reach the run state by the deadline
  • The given thread’s priority will have been dynamically increased so much by the deadline that it will have become the highest priority thread in the system