CSC501
Operating Systems Principles

Process Scheduling
Last Lecture

- Process Lifecycles
- Context Switching
Process Scheduling

Key steps

- Examine processes eligible for execution
- Select one based on a certain scheduling policy
- Switch CPU to selected item

Two-level scheduling possible

- Select Process
- Select thread within Process
Scheduling Policy

- Fundamental OS policy
- Determines when process selected for execution
- May depend on
  - Process priority
  - Time process waits
  - Process owner (user)
Process Scheduling Queues

- **Ready queue**: Set of processes residing in main memory, ready, and waiting to execute
- **Job queue**: Set of all processes in the system
- **Device queues**: Set of processes waiting for an I/O device
- **Process migration between the various queues**
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling
Schedulers

- **Long-term** scheduler (or job scheduler)
  - Which processes should be brought into the ready queue
  - Invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - Controls the degree of multiprogramming

- **Short-term** scheduler (or CPU scheduler)
  - Which process should execute next (allocates CPU)
  - Invoked very frequently (milliseconds) ⇒ (must be fast)
Schedulers (Cont.)

q Processes can be described as either:
   Q I/O-bound process
      ▶ spends more time doing I/O than computations,
      ▶ many short CPU bursts
   Q CPU-bound process
      ▶ spends more time doing computations
      ▶ few very long CPU bursts
Example Scheduling in XINU

- Each process assigned a priority
  - Non-negative integer value
  - Initialized when process created
  - Can be changed

- Scheduler chooses process with highest priority
  - Processes with the same priority are scheduled in a round-robin fashion

- Policy enforced as a system-wide invariant
The XINU Scheduling Invariant

- At any time, the CPU must run the highest priority eligible process. Among processes with equal priority, scheduling is round robin.

- Invariant enforced during
  - System call
  - Interrupt
Implementation of Scheduling

- Process eligible if state is ready or current
- To avoid searching process table
  - Keep ready processes on linked list called ready list
  - Order ready list by priority
  - Selection in constant time
Example Scheduler Code

```c
int reached()
{
    register struct pentry *optr; /* pointer to old process entry */
    register struct pentry *nptr; /* pointer to new process entry */

    /* no switch needed if current process priority higher than next*/
    if ((optr = &proctab[currid])->pstate == PRCURR) &&
        (lastkey(rdytail)<optr->pprio)) {
        return(OK);
    }

    /* force context switch */
    if (optr->pstate == PRCURR) {
        optr->pstate = PRREADY;
        insert(currpid, rdyhead, optr->pprio);
    }

    /* remove highest priority process at end of ready list */
    nptr = &proctab[currid = getlast(rdytail)];
    nptr->pstate = PRCURR;  /* mark it currently running */
    #ifdef RTCLOCK
    preempt = QUANTUM;  /* reset preemption counter */
    #endif
    ctxsw((int)&optr->pesp, (int)&optr->pirmask, (int)&nptr->pesp, (int)&nptr->pirmask);

    /* The OLD process returns here when resumed. */
    return OK;
}
```
Puzzle #1

- Invariant says that at any time, one process must be executing
- Context switch code moves from one process to another
- Question: which process executes the context switch code?
Solution to Puzzle #1

q  “Old” process
   Q  Executes first half of context switch
   Q  Is suspended

q  “New” process
   Q  Continues executing where previously suspended
   Q  Usually runs second half of context switch
Puzzle #2

- Invariant says that at any time, one process must be executing
- All user processes may be idle (e.g., applications all wait for input)
- Which process executes?
Solution to Puzzle #2

q  OS needs an extra process
  Q  Called NULL process
  Q  Never terminates
  Q  Cannot make a system call that takes it out of ready or current state
  Q  Typically an infinite loop
Lab 1 - Process Scheduling

Part I

Q The chprio() function contains a bug (sys/chprio.c)

Part II

Q The resched() function contains a limitation (sys/resched.c), i.e., a process with a lower priority could suffer from starvation
Part I

Q The chprio() function contains a bug (sys/chprio.c)

```c
/* -----------------------------------------------------------------------------
 * chprio  --  change the scheduling priority of a process
 *-----------------------------------------------------------------------------
 */
SYSCALL chprio(int pid, int newprio)
{
  STATWORD ps;
  struct pentry  *pptr;

  disable(ps);
  if (isbadpid(pid) || newprio<=0 ||
   (pptr = &proctab[pid])-&gt;pstate == PRFREE) {
    restore(ps);
    return(SYSERR);
  }
  pptr-&gt;prio = newprio;
  restore(ps);
  return(newprio);
}
```
Lab 1 - Process Scheduling

**Part II**

- The process scheduling policy has a limitation, namely process **starvation**
- You are asked to implement two different policies
  - Random scheduler
  - Proportional sharing scheduler
Random Scheduler

- Total N processes $P_i$ ($i=0..N-1$) in the ready queue
- Each $P_i$
  - Priority: $PRIO_i$
  - Probability: $PRIO_i/\text{sum}(PRIO_j)$ ($j=0..N-1$)
Proportional Sharing Scheduler

- Two factors:
  - The priority and execution time

- Timer interrupt handler
  - Related files: sys/clkint.S sys/clkinit.c
  - Interrupt rate - based on clock timer
    - ctr1000: 1ms
  - Scheduling rate:
    - Interrupt rate * QUANTUM
  - Others
    - preempt: preemption counter

More in Lecture on Interrupts
Lab 1 - Process Scheduling

q Read relevant source code in Xinu
    Q Process queue management
        v h/q.h sys/queue.c sys/insert.c, ...
    Q Proc. creation/suspension/resumption/termination:
        v sys/create.c, sys/suspend.c sys/resume.c, sys/kill.c
    Q Priority change
        v sys/chprio.c
    Q Process scheduling
        v sys/resched.c
    Q Other initialization code
        v sys/initialize.c
/* excerpt from file proc.h */

struct pentry {
    char pstate;     /* process state: PRCURR, etc. */
    int pprio;       /* process priority */
    int pesp;        /* saved stack pointer */
    STATWORD pimask; /* saved interrupt mask */
    int psem;        /* semaphore if process waiting */
    WORD pmsg;       /* message sent to this process */
    char phasmsg;    /* nonzero iff pmsg is valid */
    WORD phase;      /* base of run time stack */
    int pstklen;     /* stack length */
    WORD plimit;     /* lowest extent of stack */
    char pname[PNMLEN]; /* process name */
    int pars;        /* initial number of arguments */
    WORD paddr;      /* initial code address */
    short pdevs[2];  /* devices to close upon exit */
    int fildes[_NSFILE]; /* file - device translation */
};

extern struct pentry proctab[];
extern int numproc;     /* currently active processes */
extern int nextproc;    /* search point for free slot */
extern int currpid;     /* currently executing process */
Next Lecture

q  Process Synchronization