COL331: Operating Systems Assignment 1

Chetan Mittal

February 2019

1 System Calls

1.1 System Call Trace

1. A system call "sys_print_count" is added to syscall.c . It is implemented in sysnew.c .
2. An array syscall_count maintains the count of number of times a system call has been made since the last TRACE_ON. Whenever a system call is made with TRACE_ON; the corresponding entry in syscall_count is incremented.
3. "sys_print_count" traverses the array and prints all the entries where count > 0.

```
int
sys_print_count(void)
{
  for(int i = 1; i <= N_SYSCALLS; i++){
    if(syscall_count[i] > 0)
      cprintf("%s %d\n", syscall_name(i), syscall_count[i]);
  }
  return 0;
}
```

1.2 Toggle

1. A system call "sys_toggle" switches the trace variable.If the state is switched to TRACE_ON, it resets the syscall_count array.

```
int
sys_toggle(void)
{
  trace_on = 1 - trace_on;
  if(trace_on){
    for(int i = 0; i < N_MAXSYSCALLS; i++){
      syscall_count[i] = 0;
    }
  }
  return 0;
}
```

1.3 Add

1. A system call "sys_add" take two integer arguments and returns their sum.

```
int
sys_add(void)
{
  int input1, input2;
  if(argint(0, &input1) < 0)
    return -1;
  if(argint(1, &input2) < 0)
    return -1;
  return input1 + input2;
}
```

1.4 Process List

```
//lists all currently running processes
int
ps(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == RUNNABLE || p->state == RUNNING || p->state ==
    SLEEPING)
      cprintf("pid:%d name:%s\n", p->pid, p->name);
  release(&ptable.lock);
  return 0;
}
```

2 Inter-Process Communication

2.1 Unicast

1. Two system calls : "sys_send(int sender_pid, int rec_pid, void *msg)" and "sys_recv(void *msg)" are added.
2. Each process has its own message queue.

```
// Per-process state
struct proc {
  ...
  char name[16];               // Process name (debugging)
  char msgq[MAX_Q_LEN][MSGSIZE];// Message queue (not copied during
     fork, child process starts with empty queue)
  int qhead, qtail;             // Queue head and tail
  int rec_busy;                 // Process waiting for message
  ...
};
```

3. When a process calls "sys_send", the message is insterted in the message queue of the receiver.
4. If the receiver was waiting for a message, it is made runnable.

```
send(void)
{
  int sender_pid, rec_pid;
  char* msg;
  struct proc *p = 0;
  struct proc *temp;
  acquire(&ptable.lock);
  ... //check for valid arguments
  for(temp = ptable.proc; temp < &ptable.proc[NPROC]; temp++){
    if(temp->pid == rec_pid){
      p = temp;
    }
  }
  if((p->qtail + 1)%MAX_Q_LEN == p->qhead){
    cprintf("send: message queue full\n");
    release(&ptable.lock);
    return -4;    //message queue full
  }
  if(p->qhead == -1 && p->qtail == -1){
    p->qhead = 0;
    p->qtail = 0;
  } else {
    p->qtail = (p->qtail + 1)%MAX_Q_LEN;
  }
  memmove(p->msgq[p->qtail], msg, MSGSIZE);
  // reciever might be waiting
  wakeup1(p);
  release(&ptable.lock);
  return 0;
}
```

5. If a process calls "sys_recv", it takes out a message from its message queue.
6. If the message queue is empty, it blocks itself and waits for a message.

```
//recieve unicast message
int
recv(void)
{
  struct proc *p = myproc();
  char* msg;
  if(argptr(0, &msg, MSGSIZE) < 0)
    return -1;    //cannot read arguments
  acquire(&ptable.lock);
  while(p->qhead == -1 && p->qtail == -1){
    sleep(p, &ptable.lock);   //wait for sender
  }
  if(p->qhead == p->qtail){
    memmove(msg, p->msgq[p->qhead], MSGSIZE);
    p->qtail = -1;
    p->qhead = -1;
  } else {
    memmove(msg, p->msgq[p->qhead], MSGSIZE);
    p->qhead = (p->qhead + 1)%MAX_Q_LEN;
  }
  release(&ptable.lock);
  return 0;
}
```

2.2 Multicast

1. Two system calls "sys_send_multi(int sender_pid, int rec_pids[], void *msg)" and "set_handle(void (*handle)())" are added.
2. "set_handle" is used by a process to set the signal handler. The signal handler must be defined in the user process and have signature "void handle((void* msg)".
3. Pointer to the signal handler is stored in the process descriptor.
4. Process descriptor also stores whether a signal has been received, and also the message received.

```
// Per-process state
struct proc {
  ...
  int sig_handle_set;           // Is the signal handler set
  void (*sig_handle)(void* msg);// Signal handler
  int sig_received;             // Did the process receive a signal
  void* sig_msg;                // Message received in the last
    signal
};
```

```
//set signal handler
int set_handle(void)
{
  struct proc *curproc = myproc();
  void (*handle)(void* msg);
  if(argptr(0, (void*)&handle, sizeof(handle)) < 0)
    return -1;    //cannot read arguments
  curproc->sig_handle = handle;
  curproc->sig_handle_set = 1;
  return 0;
}
```

5. If a process does not set a signal handler yet received a signal, that signal is ignored.
6. When a process calls "sys_send_multi", sig_received of all receiving processes is set to 1 and message copied to sig_msg.

```
// p : receiving process
// called for all receiving processes
int
send_signal(int sender_pid, void* msg, struct proc* p)
{
  if(p->sig_handle_set){
    p->sig_received = 1;
    memmove(p->sig_msg, msg, MSGSIZE);
  }
  return 0;
}
```

7. When the scheduler next selects a receiving process, its signal handler is executed first with sig_msg as the argument.

```
void
scheduler(void)
{
```

```
    ...
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      if(p->sig_handle_set && p->sig_received){
        p->sig_handle(p->sig_msg);
        p->sig_received = 0;
      }
      swtch(&(c->scheduler), p->context);
      switchkvm();
      // Process is done running for now.
      c->proc = 0;
    }
    ...
}
```

8. A signal handler can do whatever the user process wants, but it typically modifies a shared variable to let the user know a signal has been received.
9. A sample signal handler is of the form :

```
volatile int num;
volatile char* msgShared;
volatile char* msgShared_temp;
//signal handler
//can't do syscall here
void sig_handler(char* msg){
  int i = MSGSIZE;
  msgShared_temp = msgShared;
  while(i--)
    *msgShared_temp++ = *msg++;
  num = 1;
  return;
}
```

10. Since the signal handler is called in user space but before switching to user stack, the return statement returns back to the kernel (scheduler).

3 Distributed Algorithm

1. A single process starts executing the distributed code (assig1_8).
2. It reads in the input arr file and stores the content in an array arr.
3. It defines the number of child processes (7), an array to store the id of each child process, and stores its id in a variable.

```
int num_child = 7;  //number of child processes
int cid[num_child]; //array to store pid of children
int block = size/num_child;
int par_id = getpid(); //parent process id
```

4. It now forks off the child processes and stores the pid of children.

```
for(int i = 0; i < num_child; i++){
  cid[i] = fork();
  if(cid[i]==0){
     // This is child
        ...
   }
}
```

5. It then waits for a message from each of the child processes giving the partial sum, and adds them up.

```
int *msg = (int*)malloc(sizeof(int));
for(int i = 0; i < num_child; i++){
    recv(msg);
    tot_sum += msg[0];
}
free(msg);
```

6. In the multicast version, it then computes the mean and waits for a dummy message from the children indicating they have set the signal handlers, and then multicasts the mean to all the child processes, and waits for the partial sums from each child process.

```
if(type==1){
    float* mean = (float*)malloc(sizeof(float));
    *mean = tot_sum/size;
    char *msg = (char*)malloc(sizeof(int));
    for(int i = 0; i < num_child; i++){
      recv(msg); //wait for dummy message
    }
    send_multi(par_id,cid,(char*)mean,num_child); //multicast mean
    char *msg = (char*)malloc(sizeof(int));
    for(int i = 0; i < num_child; i++){
        recv(msg);  //wait for partial sum
        variance += *(float*)msg;
    }
    free(mean);
    free(msg);
}
variance /= size;
```

7. Finally, it waits for all child processes to exit.

```
for(int i = 0; i < num_child; i++){
    wait();
}
```

8. Consider a child process, it computes the partial sum of arr and unicasts it to the parent process.

```
// This is child
int par_sum = 0; //partial sum
int start = i*block;
int end = (i+1)*block - 1;
if(i == num_child-1){
  end = size-1;
}
for(int j = start; j <= end; j++)
```

```
  par_sum += arr[j];
int* msg = (int*)malloc(sizeof(int));
*msg = par_sum;
send(getpid(),par_id,msg);       //unicast to parent
free(msg);
```

9. In the multicast version, the child process first sets the signal handler and sends a dummy message to the parent indicating it has set the signal handler. The signal handler is a function in the same file.

```
volatile int num;
volatile char* msgShared;
volatile char* msgShared_temp;
//signal handler
void sig_handler(char* msg){
  int i = MSGSIZE;
  msgShared_temp = msgShared;
  while(i--)
    *msgShared_temp++ = *msg++;
  num = 1;
  return;
}
```

```
if(type == 1){
  //set signal handler
  set_handle(sig_handler);
  *msg = par_sum;
  send(getpid(),par_id,msg); //dummy message to parent
  ...
}
```

10. Since the signal handler changes the shared "num" and "msg" variables, the child process repeatedly checks num to be set to 1 (depicting a signal is received)

```
while(num == 0); //waiting for signal
```

11. It now computes the partial sum of squared differences and unicasts it to the parent, and exits.

```
*mean = *(float*)msgShared;
float* sum_sq = (float*)malloc(sizeof(float));
for(int j = start; j <= end; j++)
  *sum_sq += ((float)arr[j] - *mean) * ((float)arr[j] - *mean);
send(getpid(),par_id,(char*)sum_sq);
}
exit();
```

4 Extra

4.1 Efficient implementation of queue

Queue has been implemented efficiently so that a process can receive from multiple processes at the same time. It allows for greater number of processes to share the work load and effectively communicate in a distributed algorithm. It uses locks defined in the kernel.

4.2 Efficient design for unicast

For unicast message passing, each process has been provided a separate message queue. It provides two advantages, first that it prevents data races (or the need for shared data structure) if several processes were to receive at the same time. Secondly, if a spurious process were to repeatedly send several messages to a process, a shared message buffer space might ultimately get filled up, causing all the receives to fail. But since each process has its own message queue with their own message buffers, only one process's message buffer space will get filled up, but the rest will continue to receive messages.

4.3 User locks

Locks are defined only in the kernel space. I have added locks in the user space too, so that user programs can use locks and implement shared data structures. The implementation of the user space locks is in the files lock.c and lock.h. These have also been added to the Makefile.

```
//lock.h
static inline uint
xchg(volatile uint *addr, uint newval)
{
  ...
}
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?
  char *name;        // Name of lock.
  uint cpu;   // The cpu holding the lock.
};
void            acquire(struct spinlock*, uint c);
int             holding(struct spinlock*, uint c);
void            initlock(struct spinlock*, char*);
void            release(struct spinlock*, uint c);
```

```
#Makefile
ULIB = ulib.o usys.o printf.o umalloc.o lock.o
...
EXTRA=\
  lock.c lock.h\
  ...
```